ECE 6133 Final Project Report:
C++ Rajaraman-Wong Clustering Implementation

Akshay Nagendra <akshaynag@gatech.edu>
Paul Yates <paul.maxyat@gatech.edu>

Submitted: April 26th, 2018

Introduction

During the physical design process of a digital circuit, designers and engineers must perform multiple steps to get from a design specified in a hardware-description language to the exact layout of all the various semiconductor elements for fabrication. One of the first steps in the physical design process is to partition the digital design into smaller modules to place and route the pieces of the digital circuit in real space. However, partitioning the design without taking into account the circuit complexity can result in non-optimized partitioning and placement results down the physical design process timeline. Therefore, clustering, or basically grouping gates in circuit together following certain metrics, can result in less complicated circuit structures to partition and run placement upon. While the metrics can vary, such as minimizing the number of connections between clusters, the critical delay, the cluster count, and more, the Rajaraman-Wong clustering algorithm[1] focuses on clustering while trying to minimize the critical delay in the circuit. This is to prevent the delay increase when forming the clusters but comes at the cost of area, since gates can appear in multiple clusters. Our project is a C++ implementation of the Rajaraman-Wong clustering algorithm which we call *RWClustering* and this report explores the various features and decisions during the design and implementation of *RWClustering*.

Problem Formulation

As explained previously, Rajaraman-Wong clustering involves grouping gates in a digital circuit while minimizing the increase in delay due to clustering. In addition to the input circuit, the algorithm requires a max cluster size (K), a delay specified for each node in the circuit, an inter-cluster delay (D), and a topological sorted list of all nodes in the circuit (**T**). First, a delay matrix, Δ, is generated with size N x N where N is the number of nodes in the circuit. Every entry (r_i, c_j) in the Δ matrix denotes the max delay from the output of node i to the output of node j. Once constructed, the algorithm is executed in two phases: Labeling and Clustering.

In the labeling phase, every primary input in the circuit is assigned a label equal to its delay, while every other node in the circuit gets a label value of 0. Then, for every node v in **T**, the sub-graph of v is computed and denoted as G_v. For every node x in G_v excluding v itself, a value, $l_v(x)$ is computed as the sum of x's label and $\Delta(x,v)$. These $l_v(x)$ values correspond to delays from node x to node v taking into account that the circuit is going to be clustered and involving an inter-cluster delay. Once all the $l_v(x)$ values have been computed for a node v, the largest K-1 values are added to the *cluster*(v) which holds all the nodes in v's cluster/group. Once v is also added to its own cluster, the max $l_v(x)$ value for any primary input node x in *cluster(v)* is compared with the sum of the largest $l_v(x)$ of any node x that was not added to *cluster(v)* but exists in G_v and D. The larger of these two values is picked as the label for node v and highlights

[1] R. Rajaraman and D. F. Wong, "Optimal Clustering for Delay Minimization", IEEE Trans. on Computer-Aided Design, 14(1), pp 1074-1085, 1992.

the longest delay from any topologically-prior node to node *v* whilst being conscious of the clusters being generated and added later on and how the delays would increase due to *D*.

Once all labels have been computed, the clustering phase begins where the set **L** is generated which initially holds all the primary output nodes in the circuit. While the **L** set is not empty, the first node, *n*, in **L** is removed and *cluster(n)* is formed and generated. Once formed, any nodes that have a node in *cluster(n)* as their direct successor are appended to **L** and the process continues until **L** is empty. This assures that when a cluster is formed, that all of its input clusters are also generated and formed for the final netlist result. The final result of the two phases is a clustered version of the netlist with a minimal increase in max primary input to primary output delay in the circuit. This netlist can be used for partitioning and can result in faster partitioning and placement results.

Algorithm Discussion

The core of *RWClustering* consisted of delay matrix calculation and labeling. However, to support the algorithm as well as to provide optional runtime and memory improvements, additional functionality was added including: BLIF file parsing, on-the-fly longest delay calculation, sparse matrix implementation, Lawler labeling, and non-overlapping clustering.

Each node was modeled as a data type containing an immediate previous predecessor and next successor list, label values, and an id. The id was set to the node's index in a topological ordering, and corresponded to delay matrix indices. Label values were used for clustering, and the labels were the global label and label_v (l_v) as required by Rajaraman-Wong clustering algorithm. Because the previous and next node lists contained only the immediate predecessors and successor nodes, in order to obtain a deep list of node predecessors, a recursive function was used.

Predecessor Operations

When dealing with Rajaraman-Wong clustering and related directed acyclic graph algorithms, it becomes necessary to perform an operation on all predecessors of a node, or on all predecessors which satisfy a certain condition. In these cases, the following recursive algorithm was used to locate predecessor nodes in the graph.

```
Function AddPredecessors(Node n, NodeList predecessors):
      if n has not been visited previously and satisfies conditions
            for each direct predecessor p of n
                  AddPredecessor(p, predecessors)
            set n to visited
            add n to predecessors
```

Note that there must be some visited flag for each node which must be set to false for each predecessor before this algorithm begins. In order to save on runtime and/or memory, it is wise to intelligently choose which nodes to set as unvisited. Topological sorting makes this easy, as only nodes which come topologically before a node may be its predecessor.

BLIF File Parsing

Parsing of the input BLIF files is done by using non-regular expression C++ STL library support. Each input pin, output pin, and non-IO gate is converted into a node. Since the Rajaraman-Wong clustering algorithm requires the input circuit to be a directed acyclic graph (DAG), feedback elements (i.e. latches) are split into two separate nodes. The input of the latch is converted into a primary output node and the output of the latch is converted into a primary input node. Every latch is treated as such, including latches driving other latches, to maintain the delay introduced into a circuit by latch elements and sequential logic. This results in an increase in the node count but as mentioned above, prevents feedback loops necessary for the clustering algorithm while also preserving the sequential element logic.

Delay Matrix

The delay matrix is an *N*x*N* matrix where each entry (r,c) is the longest path delay from node r to node c. If a path is not possible (reverse paths not allowed), then the delay matrix entry is 0. Delay was calculated as the delay from the output of node r to the output of node c. For example, if node r was a direct predecessor to node c, then the delay matrix entry (r,c) would simply be the delay of node c. Because nodes were topologically sorted, the delay matrix could be calculated quickly by noting all predecessors of a node would have their delay matrix entries populated first.

$$\text{max\_delay}(r,c) = \max\{\ \text{max\_delay}(r,p) + \text{delay}(c)\ \} \text{ for all } p \in \text{prev\_nodes}(c)$$

Furthermore, the lower triangular of the matrix can be set to 0 without requiring calculation time due to there being no forward path between these nodes.

This matrix causes the memory complexity of the Rajaraman-Wong clustering algorithm to be $O(N^2)$. However, the matrix will always be at least half zeroes. To improve on this, two alternate approaches were tested: on-the-fly longest path calculation and sparse matrix implementation.

On-the-fly longest path

In this extension to the base algorithm, a variation of Dijkstra's algorithm for DAGs was implemented to calculate max delay between two nodes on the fly instead of referencing a very large matrix. While longest path calculation is generally considered an NP hard travelling salesman problem, the directed acyclic and topologically sorted nature of the nodes in

Rajaraman-Wong clustering means that this problem can be solved quite quickly. This algorithm is summarized as follows

```
Function MaxDelay(Node src, Node dst):
      Let N be the set of nodes between src and dst (inclusive)
      Let array Delays be defined for each N
      Delays[N\src] = -1
      Delays[src] = 0
      for each node n in N\dst
            If Delays[n] >= 0
                  for each successor suc from n
                        if Delays[suc] < Delays[n] + delay(suc)
                              Delays[suc] = Delays[n] + delay(suc)
      MaxDelay(src,dst) = Delays[dst]
```

Despite the efficiency of this algorithm, the execution time penalty paid for using this method in *RWClustering* is prohibitively large.

Sparse matrix

In this base algorithm extension, simple sparse matrix implementation was used to decrease the memory penalty associated with using a delay matrix. This sparse matrix is a data structure which contains N rows, but each row consists only of a section of the original delay matrix row: from the first nonzero index to the last nonzero index. This is achieved by using row size and row offset values, and a dynamic array for each row. This method greatly reduced the memory required by the delay matrix while paying a small execution delay cost. Therefore, this configuration was selected as the default mode for *RWClustering* operation.

Lawler labeling algorithm

As a bonus feature of this *RWClustering* program and to provide a comparison between RW and another clustering algorithm, the Lawler labeling algorithm[2] was implemented as follows:

```
Function LawlerLabel:
      for each node n in topological order
            if n is a PI
                  label(n) = 0
            else
                  p = maximum label of n's predecessors
                  Xp = number of predecessors with label p
                  if Xp < max cluster size
                        label(n) = p
                  else
                        label(n) = p+1
```

[2] E. L. Lawler, K. N. Levitt, and J. Turner, "Module Clustering to Minimize Delay in Digital Networks", IEEE Trans. on Computers, C(18), pp 47-57, 1969.

```
Function LawlerCluster:
    For each PO node o in DAG
        For all o's predecessors n inclusive
            if label(n) is less than labels of all n's successors
                new cluster c
                l = label(n)
                for all n's predecessors p inclusive
                    if label(p) = l
                        add p to c
```

In the Lawler algorithm, nodes with the same label on the same branch form a cluster. A cluster formed around a node and its matching label predecessors only when the node's label matches none of its successors, i.e. the node is the last node on a DAG path with a certain label. It is important to note that clusters may be overlapping if there are multiple POs or reconvergent fanout.

Experimental Cluster Non-Overlap Algorithm

As an optional extension to the standard Rajaraman-Wong clustering algorithm, an experimental extension to reduce overlap between clusters was developed. As explained in the "Problem Formulation" section, Rajaraman-Wong clustering forms clusters by first forming the PO clusters and then forming clusters that are inputs to any previously formed cluster via the **L** set. By contrast, the experimental algorithm prevents overlapping clusters by flagging each node in a formed cluster. Before a cluster is formed, it must contain at least one unflagged node. In this way, when a candidate cluster contains only nodes which already exist in the formed cluster list, the candidate will be rejected, thereby reducing redundancy. Since these candidate clusters that are skipped have already been clustered, the maximum IO path delay is unaffected and therefore, the cluster count can be reduced while keeping the same maximum path delay.

Implementation Challenges

The baseline implementation of Rajaraman-Wong clustering did not encounter major obstacles once the algorithms for the implementation were developed. However, the major challenges with *RWClustering* primarily included with the extensions of Lawler labeling, "On-the-fly" delay determination, and the interactive GUI.

Lawler Labeling

In the Lawler labeling algorithm, a cluster is created for each node which has no successors which match its label. This causes cluster overlapping. An implementation was considered where nodes which belong to an existing cluster may not be added to any new cluster, but this notion was ultimately rejected due to this not being strictly faithful to the initial description of the algorithm.

On-the-fly max delay

A delay matrix requires *O(N^2)* memory complexity, while being very sparse. So a great benefit can be made if the matrix generation step can be removed from the algorithm. An attempt was made to perform on-the-fly delay matrix value calculations, i.e., calculate a max delay value between any two nodes whenever the value is required. While this max delay can be calculated quickly (*O(log(N))*) the calculation must occur between each node and all of its predecessors (*O(N^2)* times). Thus, the delay cost of this implementation is, in the majority of tested cases, less desirable than the memory penalty of using a delay matrix.

An intermediate solution could be considered. Instead of returning only one value from the max delay calculator at a time. The function could return an array of max delay values for all predecessors of the given node. The function would not pay any additional delay penalty since the function must already calculate all intermediate max delay values, but the function would be called only once per node. This solution would reduce the memory penalty to *O(N)*. This implementation is recommended for future testing.

Interactive GUI

Originally, the Qt C++ package was proposed to be used as the graphics package for *RWClustering*. However, when testing out sample Qt C++ code on the *ecelinsrv* machines, many include path errors were noticed and even after extensive debugging, the issue was no rectified, since administrator access to those machines was not provided for updating the packages. Therefore, the Python Tkinter/Turtle graphics library package was selected for building the interactive GUI since it is part of the Python 2.7 package and was installed on all *ecelinsrv* machines, except for *ecelinsrvv.ece.gatech.edu*. Although the graphics package is quick to pick up and use, the graphics library is very barebones, and after beginning the GUI implementation, it was realized that Tkinter does not have any tools that help with cross-platform integration. Therefore, the window sizes and font sizes are dependent on the OS the Python script is running on, and in order to deal with this issue, command line options for the execution script of *RWClustering*, called *RWCExecute.csh*, requires a user to specify a font preset and a resolution for the font size and resolution respectively. For more information, the READMEs in the *RWClustering/* and *RWClustering/Python* can be consulted. In hindsight, using a more sophisticated graphics package such as OpenGL or electronJS would be recommended for the GUI.

Results

Shown here are results for *RWClustering*. Figure 1 shows a general table of our results for various circuit netlist files, and the number of nodes, clusters, delay, and the execution time of the program. Figure 2 shows a more in depth breakdown for the execution times for various stages of the program for the tested circuit netlist files. Unless stated otherwise, the resulting data was compiled by executing *RWClustering* on *ecelinsrvw.ece.gatech.edu* with the max cluster size set to 8 and the inter cluster delay set to 3.[3]

Circuit Netlist	**Node Count**	**Cluster Count**	**Max IO Path Delay**	**Total Execution Time (sec)**
s9234.blif	6055	2090	88	2.223
s13207.blif	9396	2985	88	2.414
b20_opt.blif	12991	8197	103	39.001
b22_opt.blif	18789	11950	110	58.002
b17_opt.blif	25719	16147	72	74.003

Figure 1. Table showing the max delays, cluster counts, node counts, and execution times of the *RWClustering* for various circuit netlist files.

Circuit Netlist	**Topological Sorting (sec)**	**Matrix Formulation (sec)**	**Labeling Phase (sec)**	**Clustering Phase (sec)**	**Total Execution Time (sec)**
s9234.blif	0.000786	2	0.188	0.035	2.223
s13207.blif	0.001	2	0.337	0.076	2.414
b20_opt.blif	0.001	36	1	2	39.001
b22_opt.blif	0.002	52	2	4	58.002
b17_opt.blif	0.003	60	5	9	74.003

Figure 2. Table showing the various execution time lengths for the various phases of the *RWClustering* for the same circuit netlists tested in Figure 1

As can be seen in Figure 1 and Figure 2, the execution time of *RWClustering* is not only dependent on the node count (and runtime parameters), but also the nature of the circuit. One can see that the *bX_opt.blif* circuit netlists stress the application far more than the *sX.blif* circuit netlists, which is due to the complexity of the circuits. Therefore as the complexity is increased, the execution time grows more sharply as a function of node count compared to less complex circuit netlists. The overall time complexity of this clustering method is $O(N^2)$, both from the delay matrix calculation (every node forward node pair is calculated) and from the labeling section (each node must calculate the label_v for every predecessor).

[3] *RWClustering* execution times by themselves are also highly dependent on the state of the machine it is executing on. Running *RWClustering* on an Intel i7-4700MQ @ 2.4GHz with 8GB of RAM results in roughly half the execution times reported in Figure 1.

Figures 3 and 4 show the effect of the max cluster size parameter upon execution time of *RWClustering*, in addition to the maximum IO path delay from the Rajaraman-Clustering algorithm. As expected, the results indicate that as the max cluster size increases, the max IO path delay decreases since the larger the cluster size, the more gates that can be grouped together, thereby taking advantage of the intra-cluster delay in Rajaraman-Wong clustering which is assumed to be 0. However, larger clusters increase the execution time, as observable in Figure 4, since cluster generation and the clustering phase of Rajaraman-Wong clustering would involve traversal through a larger subset of nodes for every **L** set iteration. With that being said, it can be seen that the execution time penalties for cluster sizes up to 64 is within reason, although the gain in reducing the maximum IO path delay decreases with larger cluster sizes.

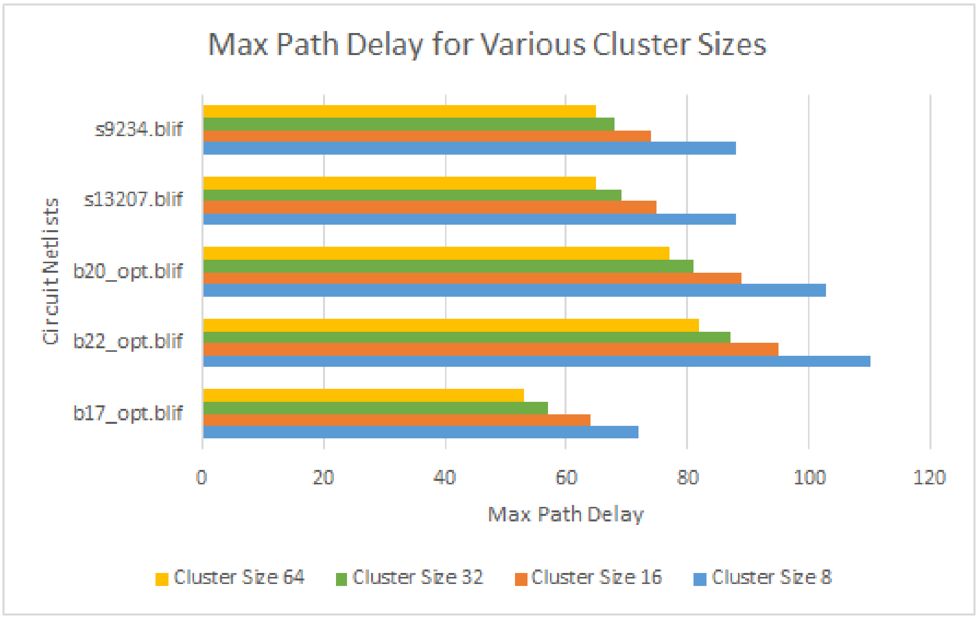


Figure 3. Bar graph depicting the max IO path delays for various input circuit files at various max cluster size limits when using *RWClustering*.

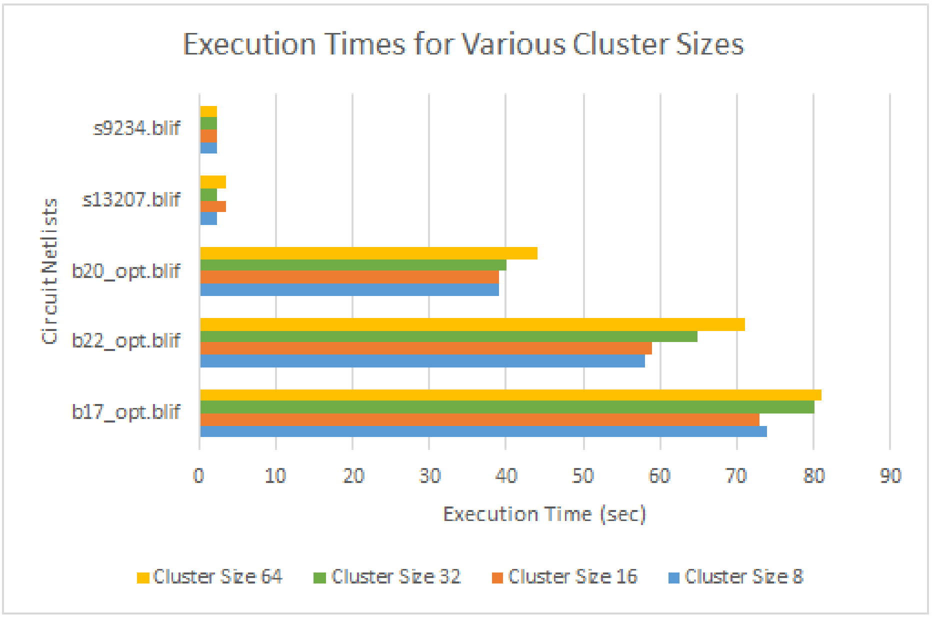


Figure 4. Bar graph depicting the execution times for various input circuit files at various max cluster size limits when using *RWClustering*.

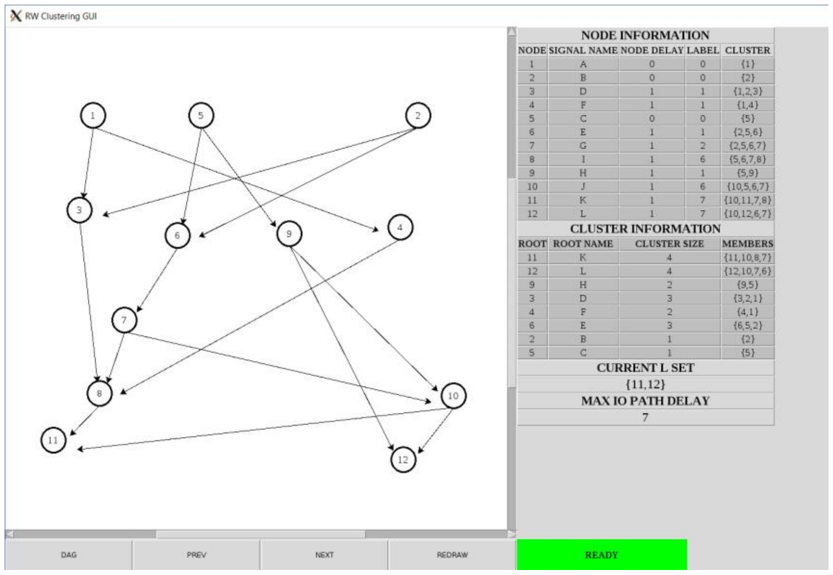
Conclusion

RWClustering offers a C++ implementation of the Rajaraman-Wong clustering algorithm with multiple features, designed for exploration of clustering problem in VLSI. The implementation allows for a reasonable execution for large circuits, but can be run with different configurations to prioritize specific goals. The strengths of *RWClustering* are the features and extended work that it supports, since it allows a user to experiment with different configurations and algorithms. Not only does *RWClustering* support a pure Rajaraman-Wong clustering implementation, but also offers an implementation of the Lawler labeling and clustering algorithm, control over the delay for each type of node, inter-cluster delay, and max cluster size, matrix configurations to prioritize either memory usage or runtime performance, and more. The base runtime performance of *RWClustering* could definitely be improved and would be the first task that would be undergone if revisited. *RWClustering* is also susceptible more so to higher degrees of complexity and interconnect within a circuit, as opposed to strictly node count, so also exploring a solution for this issue would also be a goal when revisiting this implementation.

Extended Work Results

Interactive GUI

In addition to reporting the clustering results via a verbose text file and CSV files, an interactive GUI can also be enabled for circuit netlists that sufficiently small (~ 20 gates). The GUI utilizes the Python Tkinter graphics library that allows users to create simple GUIs within a Python infrastructure. The GUI, *RWGUI*, utilizes a special file generated by *RWClustering* which contains information about nodes, labels, and clusters. *RWGUI* parses this information and constructs a DAG to be represented visually in the GUI, in addition to node and cluster information, as can be seen in Figure 5. In the interest of brevity, only the custom placement algorithm for DAG construction will be covered in this report, since the algorithm was inspired by VLSI placement algorithms. An initial placement ([x,y] coordinate pair) for each node is generated by traversing through all the nodes in a topological order and placing nodes that appear earlier in the topological order towards the top of the GUI canvas. In order to prevent the edges from crossing through other nodes and thereby affecting the readability of the DAG, a random jitter value is applied to every node's (x,y) coordinate. Once every node has a potential location, a verification function is run to make sure the overlap of edges through nodes is minimized. Once an acceptable placement has been verified, that placement is finalized and can be viewed by the user. *RWGUI* also reports how many iterations of the custom placement algorithm was required for an acceptable placement and total execution time for the algorithm.



NODE INFORMATION				
NODE	SIGNAL NAME	NODE DELAY	LABEL	CLUSTER
1	A	0	0	{1}
2	B	0	0	{2}
3	D	1	1	{1,2,3}
4	F	1	1	{1,4}
5	C	0	0	{5}
6	E	1	1	{2,5,6}
7	G	1	2	{2,5,6,7}
8	I	1	6	{5,6,7,8}
9	H	1	1	{5,9}
10	J	1	6	{10,5,6,7}
11	K	1	7	{10,11,7,8}
12	L	1	7	{10,12,6,7}

CLUSTER INFORMATION			
ROOT	ROOT NAME	CLUSTER SIZE	MEMBERS
11	K	4	{11,10,8,7}
12	L	4	{12,10,7,6}
9	H	2	{9,5}
3	D	3	{3,2,1}
4	F	2	{4,1}
6	E	3	{6,5,2}
2	B	1	{2}
5	C	1	{5}

CURRENT L SET
{11,12}

MAX IO PATH DELAY
7

Figure 5. Screenshot of *RWGUI* showing the DAG and additional information for each node, formed cluster, the max IO path delay, and the current *L* set, which the user can step through using the "PREV" and "NEXT" buttons to see how the clustering algorithm behaves.

Additional information about the GUI can be found in the *RWGUI* manual in the *docs* subdirectory in the *RWClustering* project infrastructure. As discussed in the "Implementation Issues" section, a better graphical infrastructure would be better to use such as OpenGL or electronJS, since the Python Tkinter library does not support easy scaling and has large latencies when drawing graphical elements. Currently, the issue is solved by introducing command line flags to the execution script so the user can select either a present for the font or specify the font size, as well as the resolution.

Delay Matrix Extensions Results

As discussed in the "Algorithm Discussion" section, different matrix configurations were implemented to determine the memory-latency tradeoffs for *RWClustering*. In addition to the normal *N*x*N* delay matrix from Rajaraman-Wong clustering, the sparse matrix implementation, as well as the "On-the-fly" dynamic delay determination were tested as can be seen in Figures 6 and 7. As expected, the fastest runtime was observed when utilizing the full *N*x*N* delay matrix and the slowest runtime when utilizing the "On-the-fly" delay determination. However, the memory usage for the full delay matrix hits up to 2.5GB. Therefore, for the default configuration of *RWClustering*, the sparse matrix implementation is selected since it offers the best balance between execution time and memory usage.

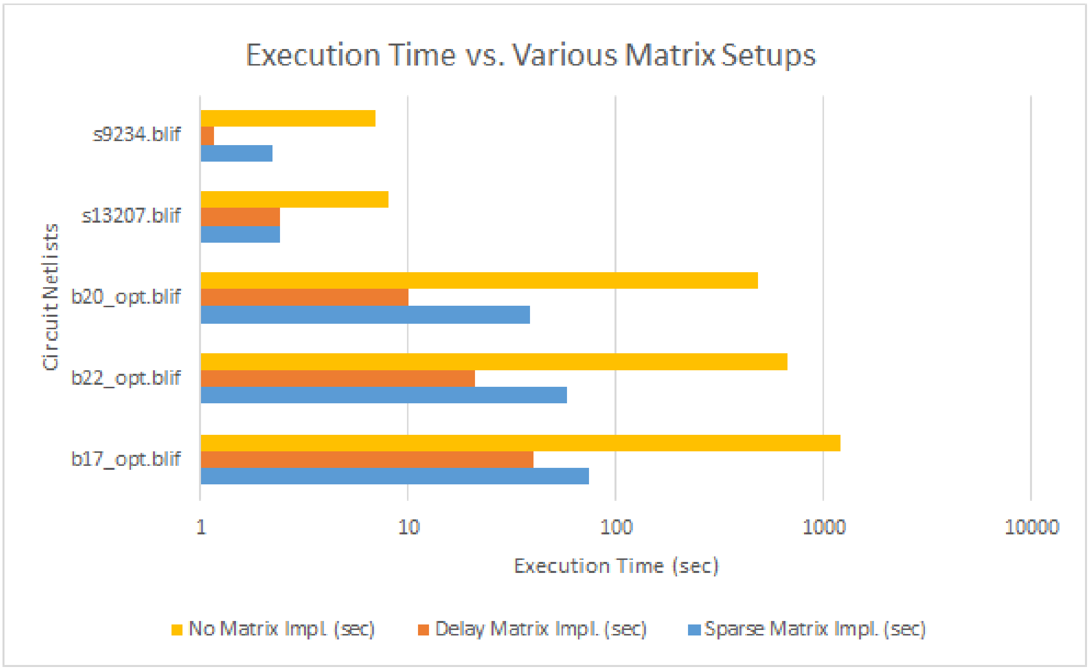


Figure 6. Bar graph depicting execution times for various matrix setups for *RWClustering*. The Sparse Matrix mode is the default mode *RWClustering* is run in since it offers a good balance between execution time and memory usage.

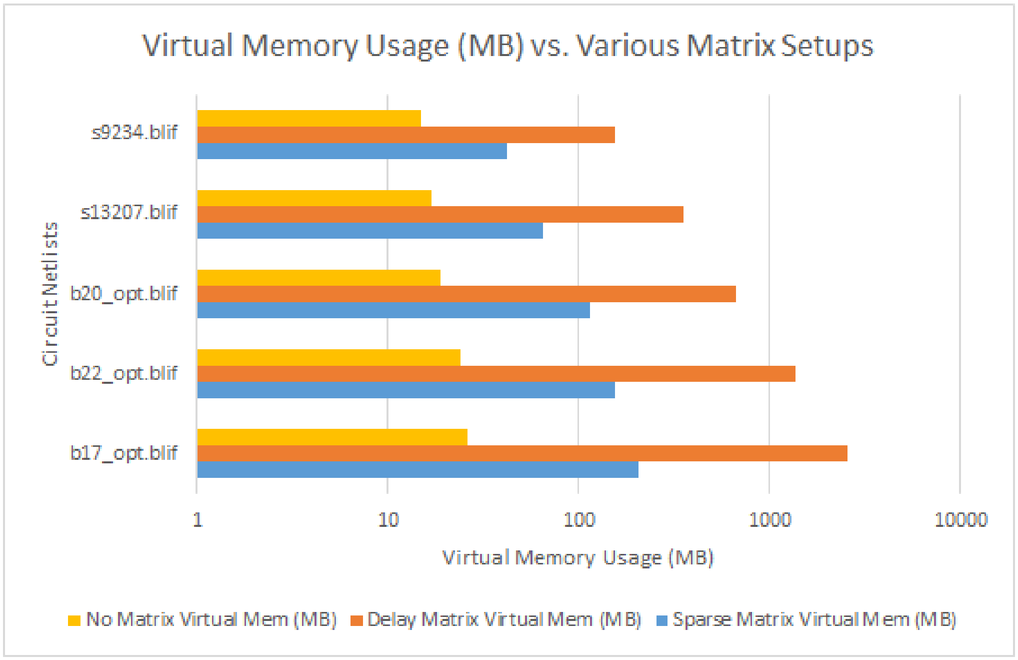


Figure 7. Bar graph depicting virtual memory usage, in MB, for various matrix setups for *RWClustering*. The Sparse Matrix mode is the default mode *RWClustering* is run in since it offers a good balance between execution time and memory usage.

Lawler Labeling Results

As discussed in the "Algorithm Discussion" section, Lawler's clustering algorithm was implemented in *RWClustering* in order to compare the Rajaraman-Wong clustering result with an additional clustering algorithm. The following results were run with the same configuration as the tests in the "Results" section, except the full *N*x*N* delay matrix implementation was configured in order to compare pure Rajaraman-Wong clustering and Lawler's clustering algorithms.

As can be seen in Figure 8, Lawler's clustering algorithm results in a clustering solution with a smaller overall cluster count and a smaller maximum IO path delay under the Unit Delay Model (UDM), where the inter-cluster delay, in addition to node delay is specified as 1. However, when converting these delay values into the General Delay Model (GDM), which is what is used in Rajaraman-Wong clustering and allows variable node and inter-cluster delay, the Lawler maximum path delays can be observed as being larger than Rajaraman-Wong clustering. This is expected since Rajaraman and Wong highlighted that one of the strengths of their algorithm was that it achieved a minimum critical delay in a clustered circuit when allowing for non-unit delays for nodes and inter-cluster delay.

Circuit Netlist	RW Cluster Count	Lawler Cluster Count	RW Max Delay	Lawler Max Delay (UDM)	Lawler Max Delay (GDM)
s9234.blif	2090	1676	88	69	99
s13207.blif	2985	2400	88	70	100
b20_opt.blif	8197	5920	103	85	118
b22_opt.blif	11950	8582	110	91	127
b17_opt.blif	16147	13518	72	56	89

Figure 8. Table highlighting the max path delay and the cluster count reported by *RWClustering* when using Rajaraman-Wong clustering (**RW**) and when using Lawler clustering (**Lawler**). The maximum path delay for Lawler clustering is reported both under the Unit Delay Model (UDM), as well as the General Delay Model (GDM).

Figure 9 shows the comparison of execution times, area costs, and memory usage between pure Rajaraman-Wong and Lawler clustering. As can be observed, since Rajaraman-Wong clustering utilizes the delay matrix, Rajaraman-Wong clustering runs faster than Lawler clustering. However, due to the smaller cluster count in the Lawler clustering solution, Lawler clustering provides a clustered netlist with a smaller increase in area when compared to the original circuit netlist. Finally, it comes as no surprise that the delay matrix in Rajaraman-Wong clustering incurs a memory usage penalty, whilst Lawler incurs a runtime penalty.

Circuit Netlist	RW Total Time (sec)	Lawler Total Time (sec)	RW Area Increase	Lawler Area Increase	RW Mem Usage (MB)	Lawler Mem Usage (MB)
s9234.blif	1.16	7.001	2.271	1.58	155	14
s13207.blif	2.433	8.001	1.909	1.491	354	16
b20_opt.blif	10.005	480.002	4.611	2.155	663	17
b22_opt.blif	21.002	540.002	4.641	2.185	1371	21
b17_opt.blif	40.002	7200	4.442	2.409	2549	22

Figure 9. Table highlighting the execution time, area increase, and memory usage reported by *RWClustering* when using Rajaraman-Wong clustering (**RW**) and when using Lawler clustering (**Lawler**). Area increase is reported as the ratio of the clustered area to the original circuit area, where each node is specified to have unit area.

Experimental Non-Overlap Extension Results

Since one of the drawbacks of the Rajaraman-Wong clustering algorithm is the overlap of clusters due to each gate potentially appearing in multiple clusters, an experimental algorithm was developed in order to reduce the cluster count, and thereby reduce the area penalty of clustering. The algorithm is executed during the clustering phase of Rajaraman-Wong clustering and prevents a candidate cluster from being formed if all of the members of that cluster have already been clustered as part of any previously formed cluster. However, this code has not been stress-tested and hence why it is referred to as "experimental," as opposed to a concrete extended feature of *RWClustering*. The results of this experimental code can be seen in Figure 10 and Figure 11 where it can be seen on average for the tested circuit netlists, the code is able to decrease the execution time of *RWClustering*, in addition to decreasing the cluster count.

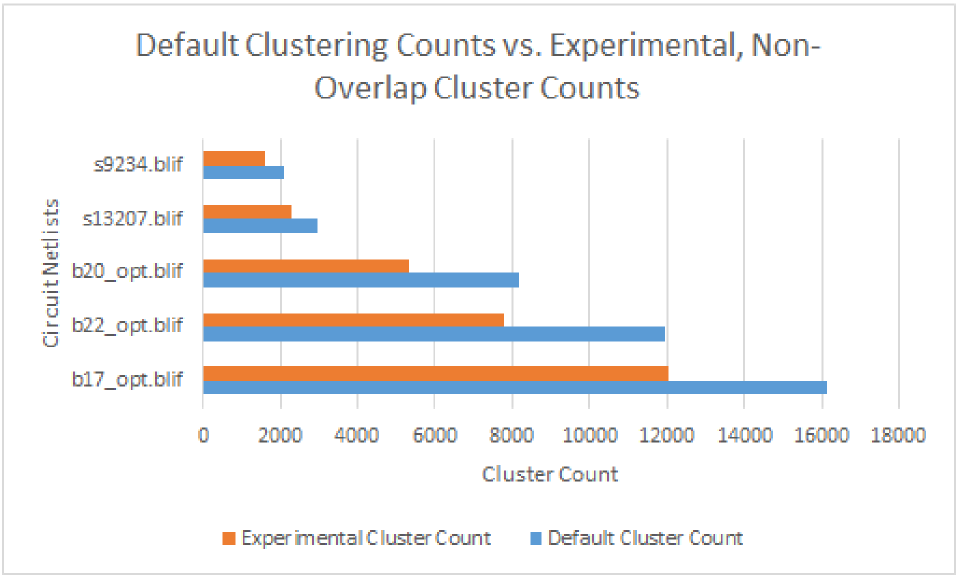


Figure 10. Bar graph highlighting the cluster counts reported by the default *RWClustering* setup and the experimental, non-overlap clustering code for various circuit netlists.

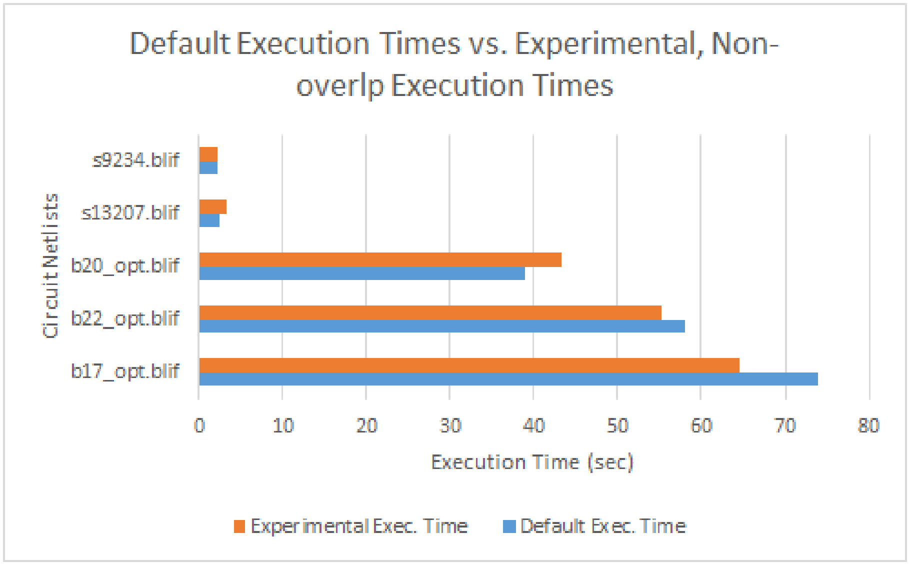


Figure 11. Bar graph highlighting the execution times reported by the default *RWClustering* setup and the experimental, non-overlap clustering code for various circuit netlists.